Machine Learning Project

Kushal Kharel

The data has been collected from accelerometers in devices like Jawbone Up, Nike FuelBand and Fitbit which are mounted on the belt, forearm, arm and dumbell of 6 participants that were performing barbell lifts correctly and incorrectly in 5 different ways.

Our goal of this project is to retrieve the data, explore the data, pre-process the data for standardization/scaling if needed and train a model to predict the manner in which participants did the exercise that is; in which of the 5 ways a barbell lift was performed. The classe variable in the data represents the 5 ways in which participants lifted barbell

The objective is to understand the manner in which participants did the exercise. For example, to lost weight; we keep track of how much calories we eat, how much we burn ,how much we burn by doing different workouts. There are several machines at the gym and we exercise on them. We try to quantify all of this activities but we never try to quantify how well we are doing it. How well we use those machines can be quantified using various techniques mentioned below.

The source data for this project comes from source: Repository

The training set comes from source: training

The testing set comes from source: testing

The dataset has been downloaded and saved in a local computer in working directory but we can also directly import the data using url.

Step - 1: Loading the data and taking the peek at the dimension of training and testing set.

```
traincsv = read.csv("C:\\Users\\kkhar\\OneDrive\\Documents\\Practical Machine Learning\\pml-training.csv
testcsv = read.csv("C:\\Users\\kkhar\\OneDrive\\Documents\\Practical Machine Learning\\pml-testing.csv")

dim(traincsv); dim(testcsv)
```

```
## [1] 19622   160
```

```
## [1]  20 160
```

Step - 2: Pre-processing the data. This is a very critical step in machine learning because our models learn from the data that gets feed into it which affects the model ability to learn. We need to keep in mind to handle Null Values, Imputation methods, standardization, how to handle ordinal and nominal factor variables, concept of One-Hot Encoding, Multi-Collinearity and its impact

To handle null values, the simple way is to drop rows and columns that contain null values. It is not always wise to drop all rows and columns that contains null values since it can result in information loss. This is where the second techniques comes in play.

Imputation is simply the process of substituting null values by using methods like Mean Imputation, Cold deck imputation, Regression imputation etc.

Standardization is the process of transforming/scaling the data such that the mean of values is zero and standard deviation is one. The formula for standardization is given below:

$$z = \frac{x_i - \mu}{\sigma}$$

where x_i is the data point and $/mu$ is the mean and σ is the standard deviation. The scale function achieves this goal in the model.

When it comes to pre-processing ordinal and nominal categorical variables, we need to treat them differently. Note that R does not use the terms nominal, ordinal and interval/ratio for types of variables. To transform ordinal categorical variables, we can use the factor function which allows us to assign an order to the nominal variables thus making it ordinal variables. We need to set the order of the parameter to TRUE and assigning a vector with the desired level of hierarchy the the argument levels.

One-Hot encoding simply means that we create 'n' columns where n is the number of unique values that the factor variable can take. Note that one-hot encoding results in multi-collinearity issues.

Multi-collinearity occurs when variables are strongly dependent on each other which can impact our model. We won't be able to use the weight vector to calculate the variable importance. To check for multi-collinearity, we plot the variables in every possible pairs (corr plot) and see the relationship between them. If they have linear relationship then variables are strongly correlated with each other and thus multi-collinearity issue. To correct this issue, we can simply drop the variables or use techniques like ridge regression or PCA or least squares regression. Simple way is to drop the variables which have VIF greater than 10.

```
# removing non-numeric variables from training set
traincsv = subset(traincsv, select = -c(X,raw_timestamp_part_1,raw_timestamp_part_2, cvtd_timestamp, ne

# removing non-numeric variables from testing set
testcsv = subset(testcsv, select = -c(X,raw_timestamp_part_1,raw_timestamp_part_2, cvtd_timestamp, new_


# creating a vector with all of the column classes from train set

columnClasses <- sapply(traincsv,class)

grep("classe", colnames(traincsv)) # getting the column number of classe variable
```

```
## [1] 153
```

```
# creating a vector with all of the column classes from test set
columnClassestest = sapply(testcsv, class)

# converting all data columns to numeric except classe variable in training set

for(i in 1:152){
  if(columnClasses[i] != "numeric"){
    if(columnClasses[i] == "factor"){
      traincsv <- traincsv[,ifelse(NAcount == 0, TRUE, FALSE)]
      traincsv[traincsv[,i] == "",i] <- NA # if there is an null record then fill it up with NA
      traincsv[,i] <- as.numeric(as.character(traincsv[,i]))
    }
    traincsv[,i] <- as.numeric(traincsv[,i])
  }
}

# converting all data columns to numeric in testing set
for(i in 1:152){
```

```
  if(columnClassestest[i] != "numeric"){
    if(columnClassestest[i] == "factor"){
      testcsv <- testcsv[,ifelse(NAcount == 0, TRUE, FALSE)]
      testcsv[testcsv[,i] == "",i] <- NA
      testcsv[,i] <- as.numeric(as.character(testcsv[,i]))
    }
    testcsv[,i] <- as.numeric(testcsv[,i])
  }
}
```

Lets count the number of NA values in training and testing set.

```
# total number of NA values in training set
library(dplyr)
traincsv %>%
  summarise(count = sum(is.na(traincsv)))
```

```
##     count
## 1 1925102
```

```
# total number of NA values in testing set
testcsv %>%
  summarise(count = sum(is.na(testcsv)))
```

```
##   count
## 1  2000
```

Removing all rows and columns with more than 80% NA in train set

```
#newtrain = traincsv[which(rowMeans(!is.na(traincsv)) > 0.80), which(colMeans(!is.na(traincsv)) > 0.80)
newtrain = traincsv[, which(colMeans(!is.na(traincsv)) > 0.80)]

# remove all columns with more than 80% NA in test set
newtest = testcsv[, which(colMeans(!is.na(testcsv)) > 0.80)]
```

```
# checking if NA's are present in the training dataset
sum(is.na(newtrain))
```

```
## [1] 0
```

```
# checking if NA's are present in the testing dataset
sum(is.na(newtest))
```

```
## [1] 0
```

Near zero variance function is used below to identify variables that have little or no variance. We do not include them in the model since it adds very little value to the algorithm. These are the variables with very few unique values relative to the number of samples and the ratio of frequency of the most common value to the frequency of the second most common value is large.

```r
#removing nerozerovariance variables from the dataset if any of them exists
nzv = nearZeroVar(newtrain[,-53], freqCut = 95/5, uniqueCut = 10, saveMetrics = TRUE, allowParallel = T
# freqCut = cutoff for the ratio of the most common value to the second most common value
# uniqueCut = cutoff for the percentage of distinct values out of the number of total samples
nzv
```

```
##                      freqRatio percentUnique zeroVar   nzv
## roll_belt             1.101904     6.7781062   FALSE FALSE
## pitch_belt            1.036082     9.3772296   FALSE FALSE
## yaw_belt              1.058480     9.9734991   FALSE FALSE
## total_accel_belt      1.063160     0.1477933   FALSE FALSE
## gyros_belt_x          1.058651     0.7134849   FALSE FALSE
## gyros_belt_y          1.144000     0.3516461   FALSE FALSE
## gyros_belt_z          1.066214     0.8612782   FALSE FALSE
## accel_belt_x          1.055412     0.8357966   FALSE FALSE
## accel_belt_y          1.113725     0.7287738   FALSE FALSE
## accel_belt_z          1.078767     1.5237998   FALSE FALSE
## magnet_belt_x         1.090141     1.6664968   FALSE FALSE
## magnet_belt_y         1.099688     1.5187035   FALSE FALSE
## magnet_belt_z         1.006369     2.3290184   FALSE FALSE
## roll_arm             52.338462    13.5256345   FALSE FALSE
## pitch_arm            87.256410    15.7323412   FALSE FALSE
## yaw_arm              33.029126    14.6570176   FALSE FALSE
## total_accel_arm       1.024526     0.3363572   FALSE FALSE
## gyros_arm_x           1.015504     3.2769341   FALSE FALSE
## gyros_arm_y           1.454369     1.9162165   FALSE FALSE
## gyros_arm_z           1.110687     1.2638875   FALSE FALSE
## accel_arm_x           1.017341     3.9598410   FALSE FALSE
## accel_arm_y           1.140187     2.7367241   FALSE FALSE
## accel_arm_z           1.128000     4.0362858   FALSE FALSE
## magnet_arm_x          1.000000     6.8239731   FALSE FALSE
## magnet_arm_y          1.056818     4.4439914   FALSE FALSE
## magnet_arm_z          1.036364     6.4468454   FALSE FALSE
## roll_dumbbell         1.022388    84.2065029   FALSE FALSE
## pitch_dumbbell        2.277372    81.7449801   FALSE FALSE
## yaw_dumbbell          1.132231    83.4828254   FALSE FALSE
## total_accel_dumbbell  1.072634     0.2191418   FALSE FALSE
## gyros_dumbbell_x      1.003268     1.2282132   FALSE FALSE
## gyros_dumbbell_y      1.264957     1.4167771   FALSE FALSE
## gyros_dumbbell_z      1.060100     1.0498420   FALSE FALSE
## accel_dumbbell_x      1.018018     2.1659362   FALSE FALSE
## accel_dumbbell_y      1.053061     2.3748853   FALSE FALSE
## accel_dumbbell_z      1.133333     2.0894914   FALSE FALSE
## magnet_dumbbell_x     1.098266     5.7486495   FALSE FALSE
## magnet_dumbbell_y     1.197740     4.3012945   FALSE FALSE
## magnet_dumbbell_z     1.020833     3.4451126   FALSE FALSE
## roll_forearm         11.589286    11.0895933   FALSE FALSE
## pitch_forearm        65.983051    14.8557741   FALSE FALSE
## yaw_forearm          15.322835    10.1467740   FALSE FALSE
## total_accel_forearm   1.128928     0.3567424   FALSE FALSE
## gyros_forearm_x       1.059273     1.5187035   FALSE FALSE
## gyros_forearm_y       1.036554     3.7763735   FALSE FALSE
## gyros_forearm_z       1.122917     1.5645704   FALSE FALSE
## accel_forearm_x       1.126437     4.0464784   FALSE FALSE
```

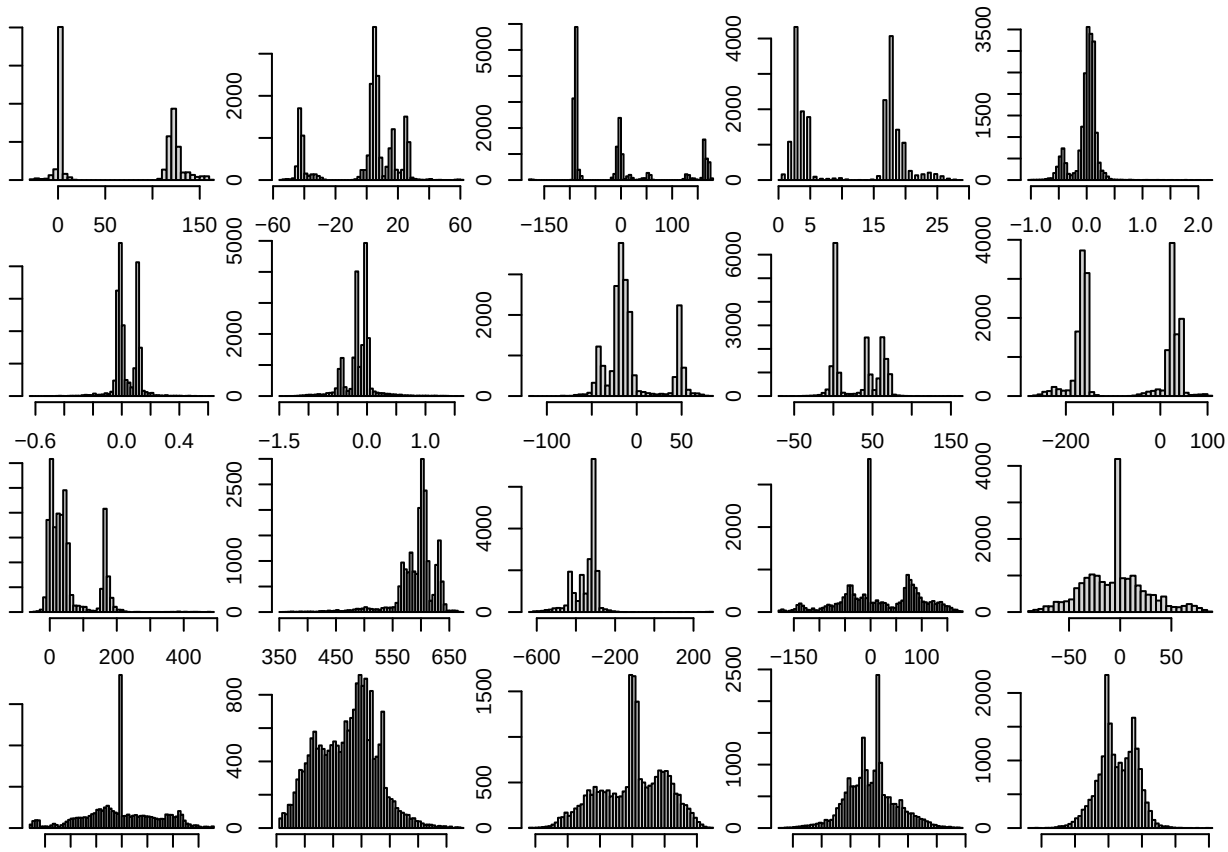
```
## accel_forearm_y       1.059406     5.1116094   FALSE FALSE
## accel_forearm_z       1.006250     2.9558659   FALSE FALSE
## magnet_forearm_x      1.012346     7.7667924   FALSE FALSE
## magnet_forearm_y      1.246914     9.5403119   FALSE FALSE
## magnet_forearm_z      1.000000     8.5771073   FALSE FALSE
```

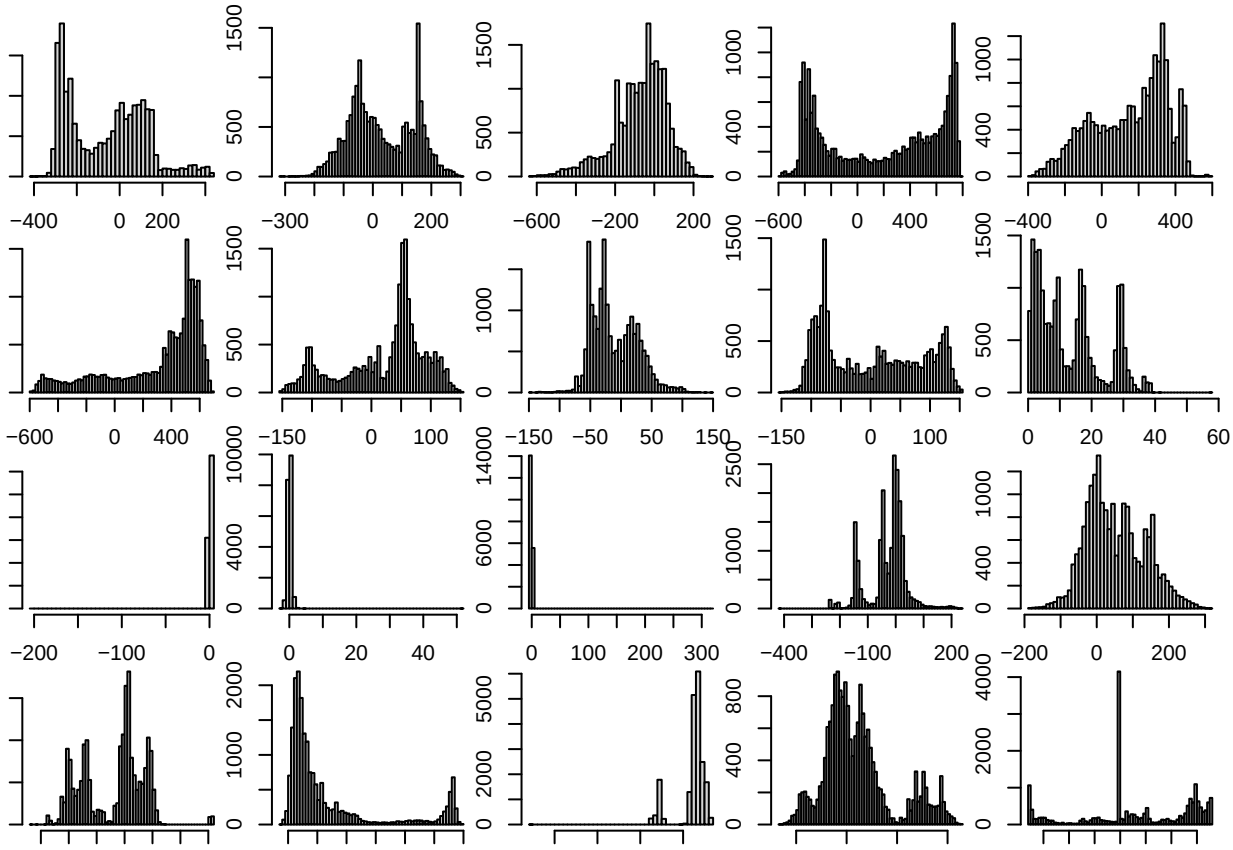
Since none of the variables has nzv true, we are including all of the variables in our model.

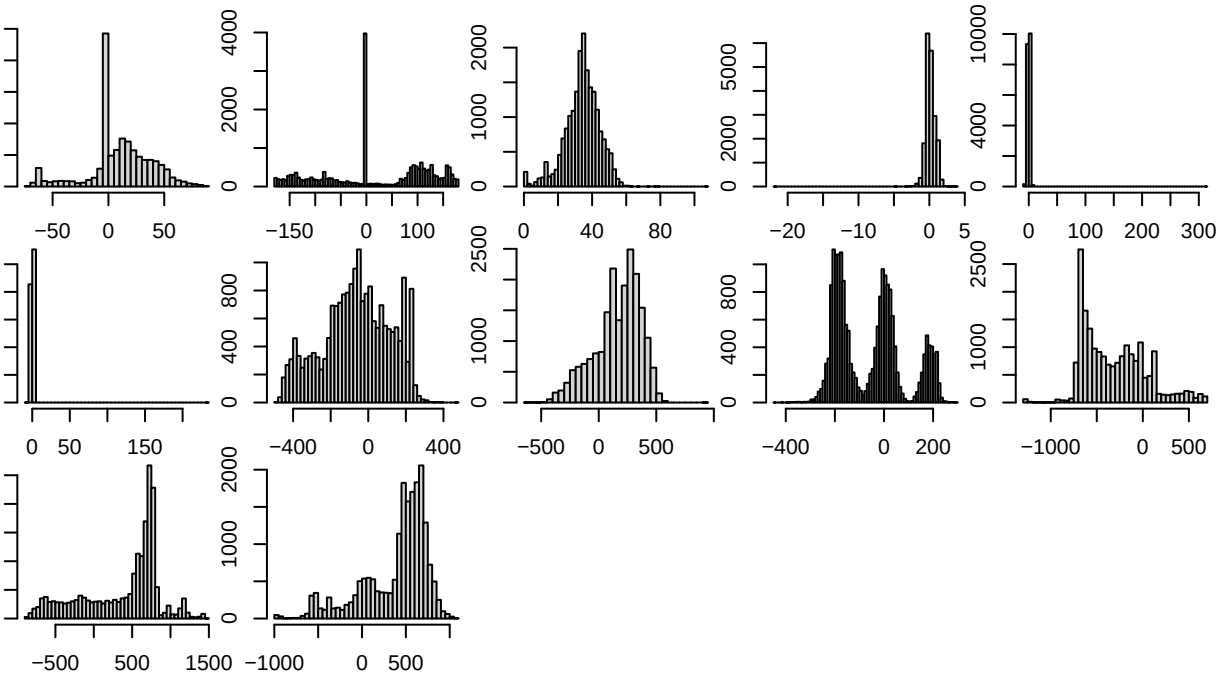
Step 3: Checking the distribution of the data

Lets plot the histogram of all of the independent variables to see how the data in each of them is being distributed.

```
# looking at the histogram of data
library(Hmisc)

par(mar=c(1,1,1,1))

Hmisc::hist.data.frame(newtrain[-53])
```




0 500 1500
−400 0 200
0 500 1500
−300 0 200
0 500 1500
−600 −200 200
0 400 1000
−600 0 400
0 400 1000
−400 0 400
0 500 1500
−600 0 400
0 500 1500
−150 0 100
0 1000
−150 −50 50 150
0 500 1500
−150 0 100
0 500 1500
0 20 40 60
0 4000 10000
−200 −100 0
0 6000 14000
0 100 200 300
0 1000 2500
−400 −100 200
0 400 1000
−200 0 200
0 1000 2000
−200 −100 0
0 2000 5000
0 20 40
0 400 800
−400 −100 200
0 2000 4000
−200 0 200



Step 4: Partitioning the dataset

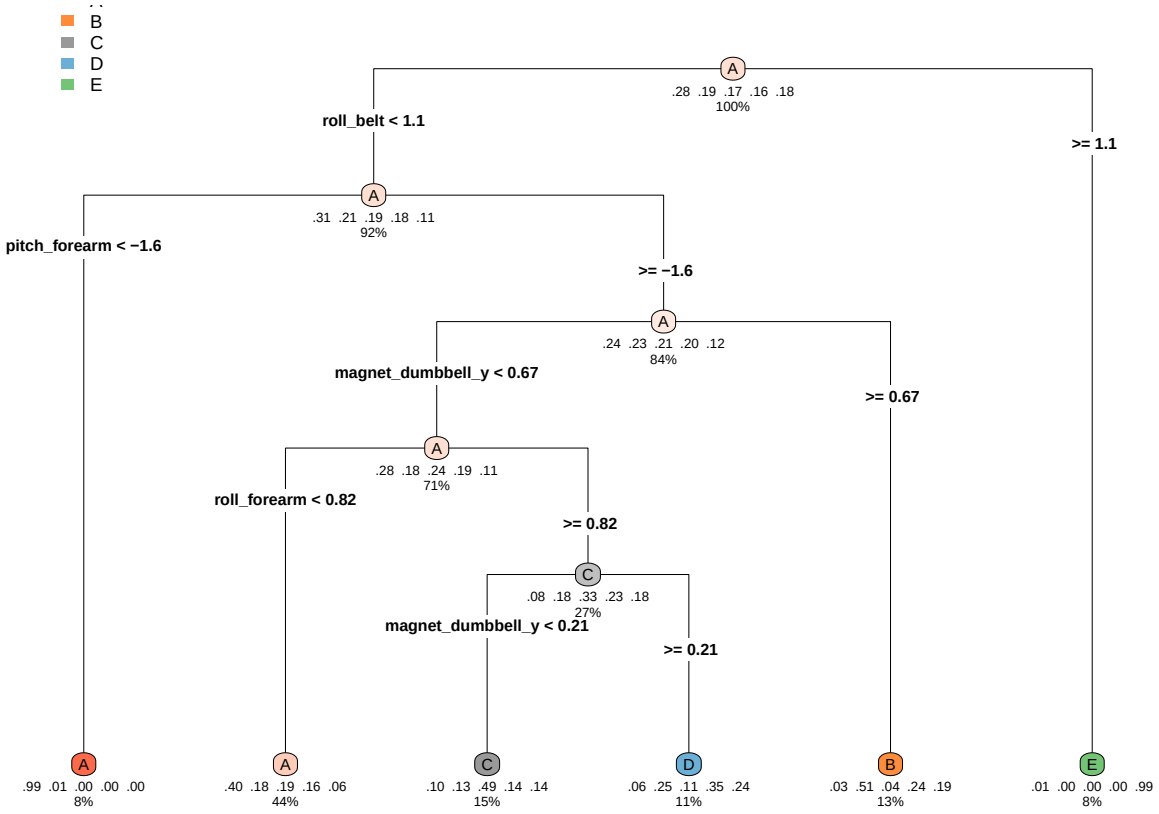
```
# dividing the training set into validation set, sub-training set and testing set

inBuild = createDataPartition(y = newtrain$classe, p = 0.75, list = FALSE)

validation = newtrain[-inBuild,]; buildData = newtrain[inBuild, ]
inTrain = createDataPartition(y = buildData$classe, p = 0.75, list = FALSE)

training = buildData[inTrain, ]
testing = buildData[-inTrain, ]
```

Step 5: Creating the model, Training the model, Testing the Model and Validating the Model

First Model: Decision Trees: Decision Tree creates classification or regression models in tree structure. It breaks down a dataset into smaller subsets with increasing depth of tree with leaf and decision nodes. The decision node has two or more branches and leaf node represents a classification or decision. The root node corresponds to the top-most node in a tree which is the best predictor. Selecting the top-most node is out of the scope here. Note: Decision trees can handle categorical and numerical data.

```
modelFit1 = train(classe~., data = training,
                  preProcess = c("center", "scale"), method = "rpart",
                  trControl = trainControl(method="cv", number=3, verboseIter=F), tuneLength = 4) # sett

# Note: Standardization within the training process.
```

```
par(mfrow=c(1,1))

# Plotting decision tree
rpart.plot::rpart.plot(modelFit1$finalModel, cex = 0.5, type = 4, under = TRUE)
```




```
prediction1 = predict(modelFit1, testing)
cmdecisiontree <- confusionMatrix(prediction1, factor(testing$classe))
cmdecisiontree
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   A   B   C   D   E
##          A 943 288 297 301 106
##          B  21 251  24  96  90
##          C  56  89 267  89  85
##          D  23  84  53 117 100
##          E   3   0   0   0 295
##
## Overall Statistics
##
##                Accuracy : 0.5092
##                  95% CI : (0.493, 0.5255)
##     No Information Rate : 0.2844
##     P-Value [Acc > NIR] : < 2.2e-16
```

```
##
##                   Kappa : 0.3589
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.9015  0.35253  0.41654  0.19403  0.43639
## Specificity            0.6231  0.92212  0.89496  0.91545  0.99900
## Pos Pred Value         0.4873  0.52075  0.45563  0.31034  0.98993
## Neg Pred Value         0.9409  0.85576  0.87904  0.85277  0.88728
## Prevalence             0.2844  0.19358  0.17428  0.16395  0.18380
## Detection Rate         0.2564  0.06824  0.07259  0.03181  0.08021
## Detection Prevalence   0.5261  0.13105  0.15933  0.10250  0.08102
## Balanced Accuracy      0.7623  0.63732  0.65575  0.55474  0.71770
```

In the decision tree plot, we start at the root node (depth 0): At the top, it is the overall probability of participants that are in A. 28% from classe A are classified as A (this is also a prevalance in statistics summary), 19% from B are classified as A, 17% from C are classified as A, 16% from D are classified as A and 18% from E are classified as A. The node asks whether roll belt is less than 1.1 or not. If yes, then we go down to the root's left child node. Further, the node asks if pitch forearm is less than -1.6. If yes, it classifies the classe as A. We keep on going to understand what impacts the likelihood of falling into one of the given classe.

From the results above, decision tree is not able to accurately classify the classe. The accuracy rate is around 53%, it can change everytime we run the model but should be close to this number. The model was able to classify classe A pretty well but not others. We can try to fit a different model and see how it performs.

Second model: Random Forest: Random Forest is the ensemble of decision trees. It combines multiple decision trees to get more accurate predictions. Random forest chooses the predictors at random and takes the outputs of multiple trees to make a decision. Think about decision trees and random forests as individual work and group work.

```
# random forest
library(randomForest)
modelFit2 = randomForest(as.factor(training$classe)~., data = training, ntree=10, importance = T
                         # preProcess = c("center","scale"),
                         # methods = "rf",
                         # trControl = trainControl(method="cv", number=3, verboseIter=F),  # setting u
                                     # importance = TRUE, proximity = TRUE
                                     )
modelFit2
```

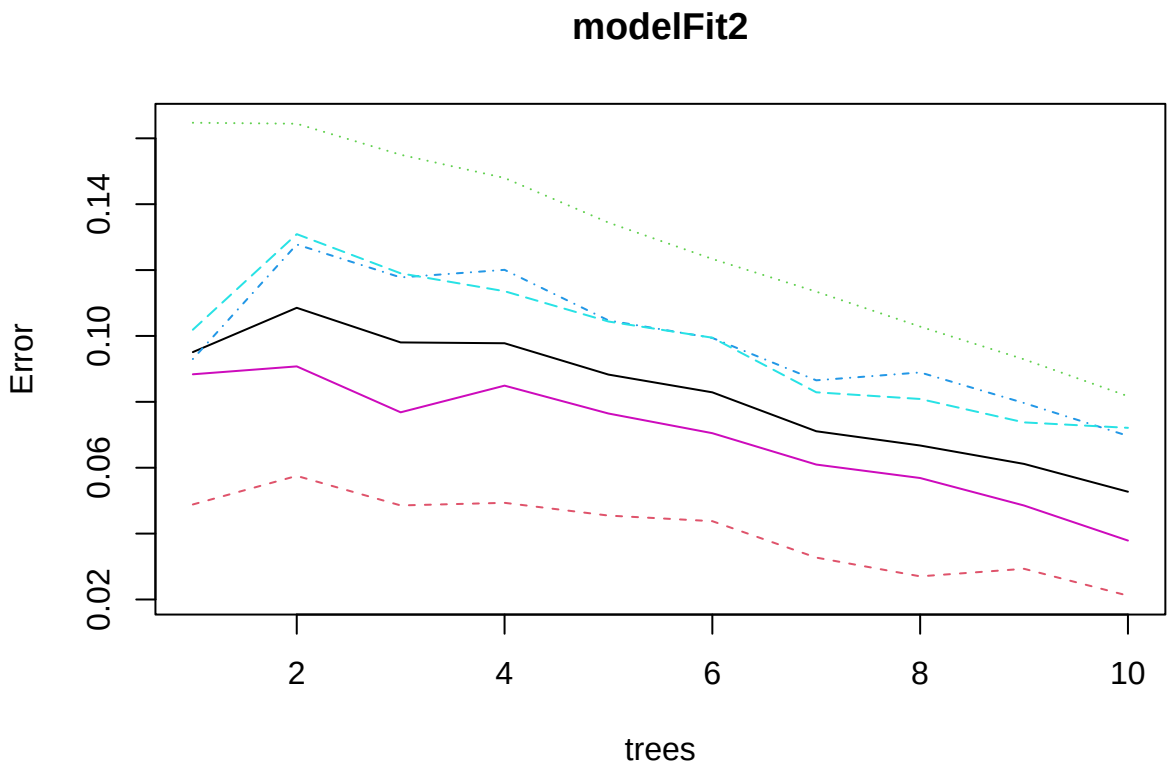
```
##
## Call:
##  randomForest(formula = as.factor(training$classe) ~ ., data = training,      ntree = 10, importance
##                Type of random forest: classification
##                      Number of trees: 10
## No. of variables tried at each split: 7
##
##         OOB estimate of  error rate: 5.27%
## Confusion matrix:
##      A    B    C    D    E class.error
## A 3052   33    9   15    9  0.02116742
```

```
## B   79 1933   47   24   22  0.08171021
## C   11   74 1776   37   11  0.06966998
## D   19   30   67 1660   13  0.07210732
## E    6   29   19   22 1930  0.03788634
```

```
#getTree(modelFit2, k = 4)
```

We can see from training the random forest model, there are 10 trees and number of variables tried at each split is seven. From the confusion matrix, we can see that 3026 were correctly identified as A which is indeed A. 77 are classified as A but are B and so on.

```
plot(modelFit2)
```



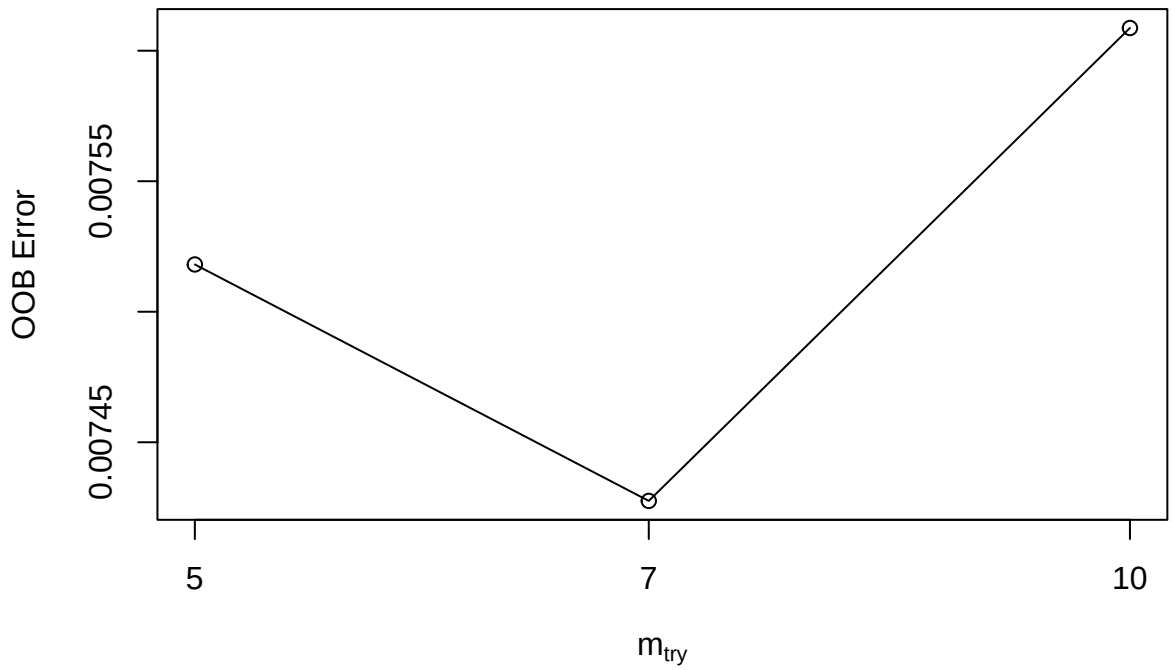
From the model fit plot, we can see that as the number of trees increases, the error decreases

We can also tune the model and make it more accurate by tweaking various hyper parameters. One simple example of hyperparameter tuning is below.

```
# tuning the model

# ntreeTry: The number of trees to build.
# mtryStart: The starting number of predictor variables to consider at each split.
# stepFactor: The factor to increase by until the out-of-bag estimated error stops
# improving by a certain amount.
# improve: The amount that the out-of-bag error needs to improve by to
# keep increasing the step factor
```

```r
# hyper parameter tuning
tuned_model = tuneRF(
  x= training[,-53], #define predictor variables
  y=as.factor(training$classe), #define response variable
  ntreeTry=500,
  stepFactor=1.5,
  improve=0.01,
)
```

```
## mtry = 7  OOB error = 0.74%
## Searching left ...
## mtry = 5     OOB error = 0.75%
## -0.01219512 0.01
## Searching right ...
## mtry = 10    OOB error = 0.76%
## -0.02439024 0.01
```




```r
print(tuned_model)
```

```
##        mtry    OOBError
## 5.OOB     5 0.007518116
## 7.OOB     7 0.007427536
## 10.OOB   10 0.007608696
```

We can see the number of predictors used at each split when building the trees on the x-axis and the out-of-bag estimated error on the y-axis. The lowest OOB error is achieved by using 7 randomly chosen predictor at each split when building the trees which is default in the function above.

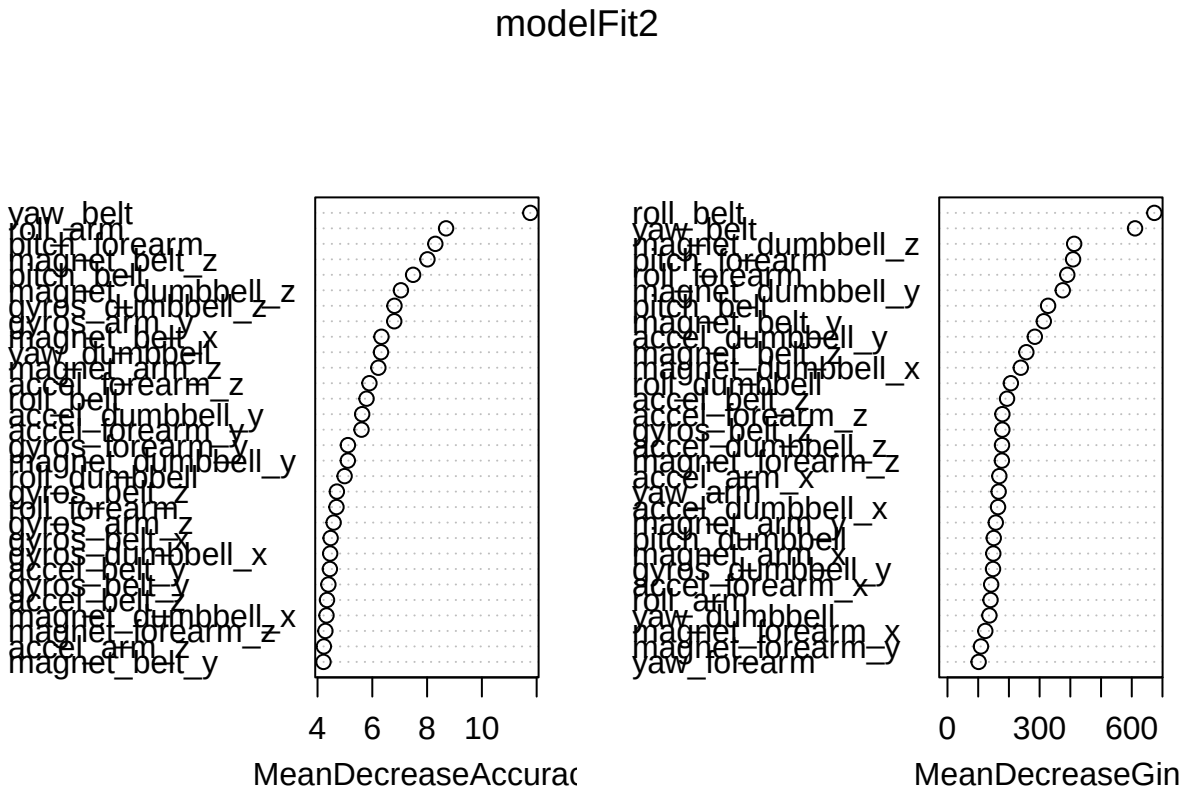
```
prediction2 = predict(modelFit2, testing)
cmrandomforest = confusionMatrix(prediction2, factor(testing$classe))
cmrandomforest
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    A    B    C    D    E
##          A 1043    9    0    0    0
##          B    1  698   11    1    1
##          C    0    4  627    6    1
##          D    1    1    3  595    2
##          E    1    0    0    1  672
##
## Overall Statistics
##
##                Accuracy : 0.9883
##                  95% CI : (0.9843, 0.9915)
##     No Information Rate : 0.2844
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9852
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.9971   0.9803   0.9782   0.9867   0.9941
## Specificity            0.9966   0.9953   0.9964   0.9977   0.9993
## Pos Pred Value         0.9914   0.9803   0.9828   0.9884   0.9970
## Neg Pred Value         0.9989   0.9953   0.9954   0.9974   0.9987
## Prevalence             0.2844   0.1936   0.1743   0.1639   0.1838
## Detection Rate         0.2836   0.1898   0.1705   0.1618   0.1827
## Detection Prevalence   0.2860   0.1936   0.1735   0.1637   0.1833
## Balanced Accuracy      0.9969   0.9878   0.9873   0.9922   0.9967
```

From the Confusion Matrix and Statistics summary above, we can see that random forest predicted 1043 records as A which were indeed A. It was able to accurately identify the classe 98.69%.

Variance importance plot helps us identify which predictor variables are important in the final model aka checking for distributional assumptions. The presence of variance in data set is important because it allows the model to learn about different patterns hidden in the data. Also, it must not have high variance because then we will be over-fitting the model. We need a balance between these two. Also refer to bias-variance trade-off to learn more.

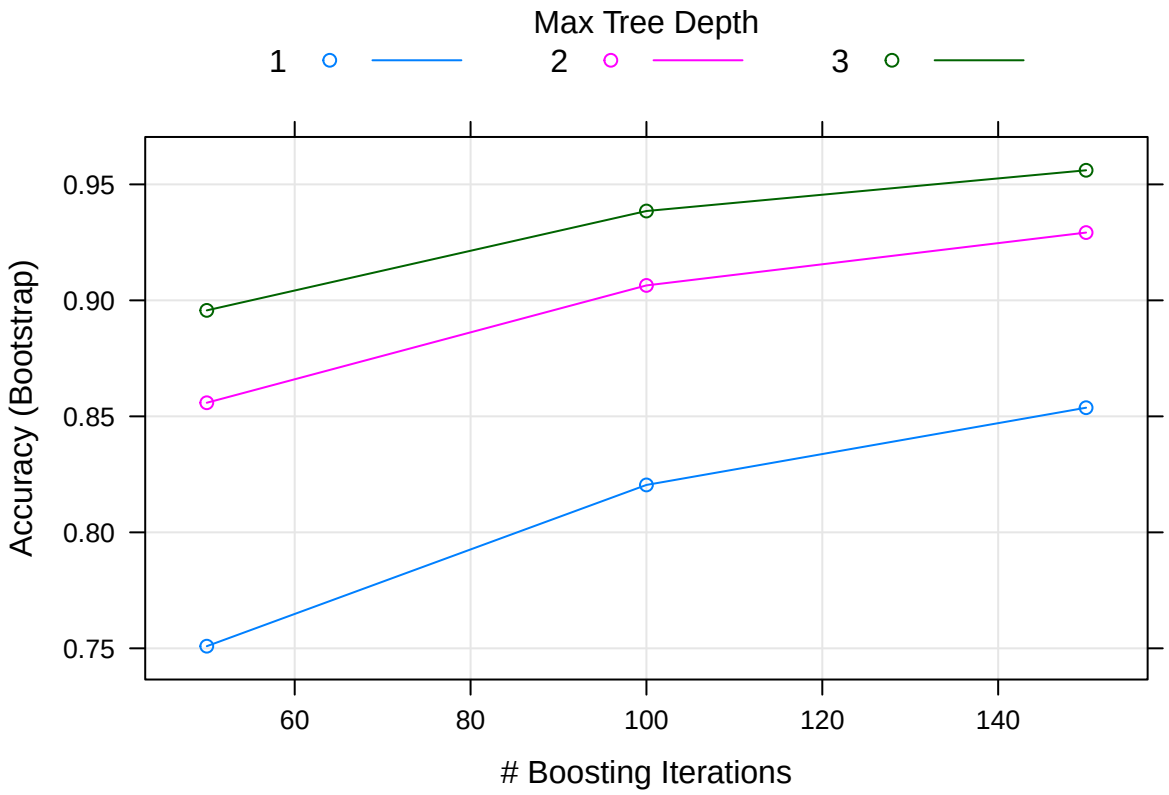
```
varImpPlot(modelFit2,
           type=NULL, class=NULL, scale=TRUE,
           main=deparse(substitute(modelFit2)))
```



The x-axis displays the average increase in node purity of the regression trees based on splitting on the various predictors displayed on the y-axis. From the plot we can see that yaw_belt and roll_belt are the most important predictor variable,

Third Model: Gradient Boosting: This is one of the leading methods to win Kaggle competitions. On one hand, random forest builds ensemble of independent trees whereas on the other hand, GBM builds ensemble of shallow and weak successive trees with each tree learning and improving on the previous. When we combine these weak trees, it can often produce hard to beat predictions than other algorithms can. Gradient boosting is considered gradient descent algorithm which is an optimization algorithm capable of finding optimal solutions to a given problem. The general idea is to tweak the parameters iteratively to minimize the cost function. The problem is that not all cost functions are convex, we might get stuck in local minima which can make finding global minima difficult.

```
# Gradient Boosted Trees

modelFit3 = train(classe~., method = "gbm", data = training,
                  preProcess = c("center", "scale"),
                  verbose = FALSE) #boosting with trees

plot(modelFit3)
```




```
prediction3 = predict(modelFit3, testing)

cmboostingtrees = confusionMatrix(prediction3, as.factor(testing$classe))
cmboostingtrees
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    A    B    C    D    E
##          A 1024   23    0    0    0
##          B   10  657   23    3    7
##          C    4   24  608   21    2
##          D    5    8    7  577    6
##          E    3    0    3    2  661
##
## Overall Statistics
##
##                Accuracy : 0.9589
##                  95% CI : (0.952, 0.9651)
##     No Information Rate : 0.2844
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.9481
##
##  Mcnemar's Test P-Value : 9.8e-05
```

```
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.9790   0.9228   0.9485   0.9569   0.9778
## Specificity            0.9913   0.9855   0.9832   0.9915   0.9973
## Pos Pred Value         0.9780   0.9386   0.9226   0.9569   0.9880
## Neg Pred Value         0.9916   0.9815   0.9891   0.9915   0.9950
## Prevalence             0.2844   0.1936   0.1743   0.1639   0.1838
## Detection Rate         0.2784   0.1786   0.1653   0.1569   0.1797
## Detection Prevalence   0.2847   0.1903   0.1792   0.1639   0.1819
## Balanced Accuracy      0.9851   0.9541   0.9659   0.9742   0.9876
```

We can see from the results above, gradient boosting was not a great choice against random forest but it is better than decision trees.The accuracy is 95.68%

Fourth Model: Support Vector Machines: SVM is widely used in classification problems. SVM creates a decision boundary which best separates the given data points. The best hyperplane is the one whose distance to the nearest element of each tag is the largest. Each data item is plotted as a point in n-dimensional space where n is the number of independent variables with the value of each independent variables being the value in particular coordinate. SVM can efficiently perform a non-linear classification, by implicitly mapping the inputs into high-dimensional variable spaces.

```
# Support Vector Machine

modelFit4 = train(classe~., method = "svmLinear",
                  data = training,
                  preProcess = c("center", "scale"),
                  trControl = trainControl(method="cv", number=3, verboseIter=F),  # setting up control
                  tuneLength = 4)

prediction4 = predict(modelFit4, testing)

cmsupportvector = confusionMatrix(prediction4, factor(testing$classe))
cmsupportvector
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   A   B   C   D   E
##          A 949 104  62  37  37
##          B  30 505  49  35  87
##          C  29  43 494  70  48
##          D  33  11  21 443  34
##          E   5  49  15  18 470
##
## Overall Statistics
##
##                Accuracy : 0.7779
##                  95% CI : (0.7641, 0.7912)
##     No Information Rate : 0.2844
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7176
```

```
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.9073   0.7093   0.7707   0.7347   0.6953
## Specificity            0.9088   0.9322   0.9374   0.9678   0.9710
## Pos Pred Value         0.7981   0.7153   0.7222   0.8173   0.8438
## Neg Pred Value         0.9610   0.9303   0.9509   0.9490   0.9340
## Prevalence             0.2844   0.1936   0.1743   0.1639   0.1838
## Detection Rate         0.2580   0.1373   0.1343   0.1204   0.1278
## Detection Prevalence   0.3233   0.1920   0.1860   0.1474   0.1514
## Balanced Accuracy      0.9080   0.8208   0.8541   0.8512   0.8331
```

From the results above, SVM performed worse than random forest and gradient boosted trees but still better than decision tree. The accuracy is only 77.98%

Let's combine two worst performing model and create a new model to see whether it performs better than all of the model we have trained so far.

Ensemble Method (combining two methods that have low accuracy, model stacking). In our model, we are stacking the predictions together using random forests and setting up control for training to use 3-fold cross-validation.

```
comb = data.frame(prediction1, prediction4, classe = as.factor(testing$classe))
modelComb = randomForest(as.factor(comb$classe)~., data = comb,
                         preProcess = c("center,scale"),
                         trControl = trainControl(method="cv", number=3, verboseIter=F),
                         importance = TRUE, proximity = TRUE)

getTree(modelComb, k = 4)
```

```
##    left daughter right daughter split var split point status prediction
## 1              2              3         2           1      1          0
## 2              0              0         0           0     -1          1
## 3              4              5         1          15      1          0
## 4              6              7         2           8      1          0
## 5              0              0         0           0     -1          5
## 6              0              0         0           0     -1          4
## 7              8              9         2           4      1          0
## 8              0              0         0           0     -1          3
## 9             10             11         2           2      1          0
## 10             0              0         0           0     -1          2
## 11            12             13         1           5      1          0
## 12             0              0         0           0     -1          5
## 13            14             15         1           2      1          0
## 14             0              0         0           0     -1          5
## 15             0              0         0           0     -1          5
```

```
predictionComb = predict(modelComb, testing)
cmensemble = confusionMatrix(predictionComb, comb$classe)
cmensemble
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   A   B   C   D   E
##          A 946 104  62  37  23
##          B  30 507  50  36  78
##          C  29  41 493  69  29
##          D  33  11  21 443  30
##          E   8  49  15  18 516
##
## Overall Statistics
##
##                Accuracy : 0.7898
##                  95% CI : (0.7763, 0.8029)
##     No Information Rate : 0.2844
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7329
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.9044   0.7121   0.7691   0.7347   0.7633
## Specificity            0.9141   0.9346   0.9447   0.9691   0.9700
## Pos Pred Value         0.8072   0.7233   0.7458   0.8234   0.8515
## Neg Pred Value         0.9601   0.9311   0.9509   0.9490   0.9479
## Prevalence             0.2844   0.1936   0.1743   0.1639   0.1838
## Detection Rate         0.2572   0.1378   0.1340   0.1204   0.1403
## Detection Prevalence   0.3187   0.1906   0.1797   0.1463   0.1648
## Balanced Accuracy      0.9093   0.8233   0.8569   0.8519   0.8667
```

Let us check the accuracy of each model in the test set.

```
# What is the resulting accuracy on the test set?
# Is it better or worse than each of the individual predictions?

confusionMatrix(prediction1, as.factor(testing$classe))$overall[1]
```

```
##  Accuracy
## 0.5092442
```

```
confusionMatrix(prediction2, as.factor(testing$classe))$overall[1]
```

```
##  Accuracy
## 0.9883089
```

```
confusionMatrix(prediction3, as.factor(testing$classe))$overall[1]
```

```
##  Accuracy
## 0.9589451
```

```
confusionMatrix(prediction4, as.factor(testing$classe))$overall[1]
```

```
##  Accuracy
## 0.7778684
```

```
confusionMatrix(predictionComb, as.factor(testing$classe))$overall[1]
```

```
##  Accuracy
## 0.7898314
```

We can see from the results above that the combined method has much less error than the individual models. However, random forest is still the best performing model among all of them.

We choose random forest model as the most appropriate model for this data.

Now let's apply the prediction to validation set.

```
# Prediction on validation data set
pred1v = predict(modelFit1, validation);
pred2v = predict(modelFit2, validation);
pred3v = predict(modelFit3, validation);
pred4v = predict(modelFit4, validation);

predvdf = data.frame(prediction1 = pred1v, prediction2 = pred2v,
                     prediction3 = pred3v, prediction4 = pred4v)

combpredv = predict(modelComb,predvdf)
```

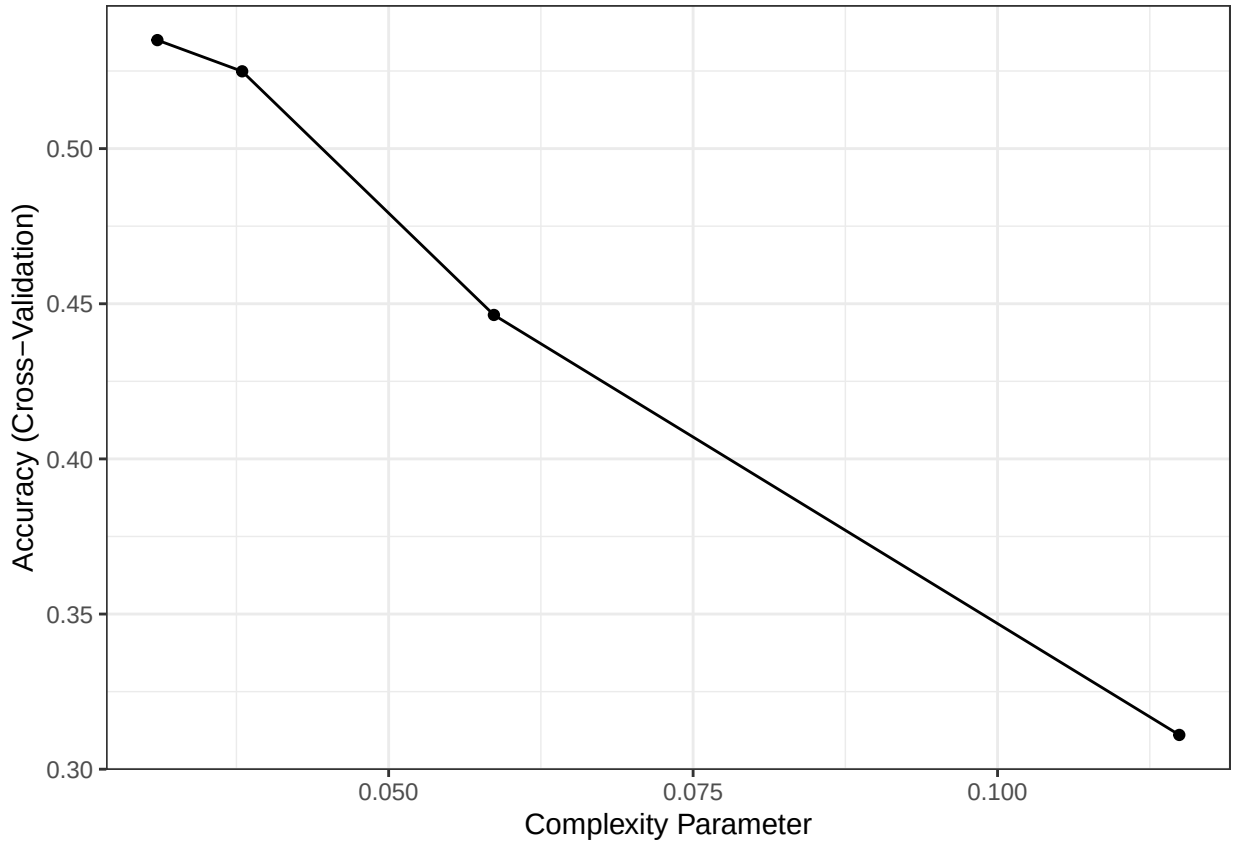
Evaluation:

```
# Evaluate on Validation
cmdecisiontreev = confusionMatrix(pred1v, as.factor(validation$classe))$overall[1]
cmrandomforestv = confusionMatrix(pred2v, as.factor(validation$classe))$overall[1]
cmboostingtreev = confusionMatrix(pred3v, as.factor(validation$classe))$overall[1]
cmsupportvectorv = confusionMatrix(pred4v, as.factor(validation$classe))$overall[1]
cmensemblev = confusionMatrix(combpredv, as.factor(validation$classe))$overall[1]

models <- c("Tree", "RF", "GBM", "SVM", "Ensemble")
accuracy <- round(c( cmdecisiontreev, cmrandomforestv,
                     cmboostingtreev, cmsupportvectorv,
                     cmensemblev),3) #accuracy

oos_error <- 1 - accuracy #out of sample error

data.frame(accuracy = accuracy, oos_error = oos_error, row.names = models)
```

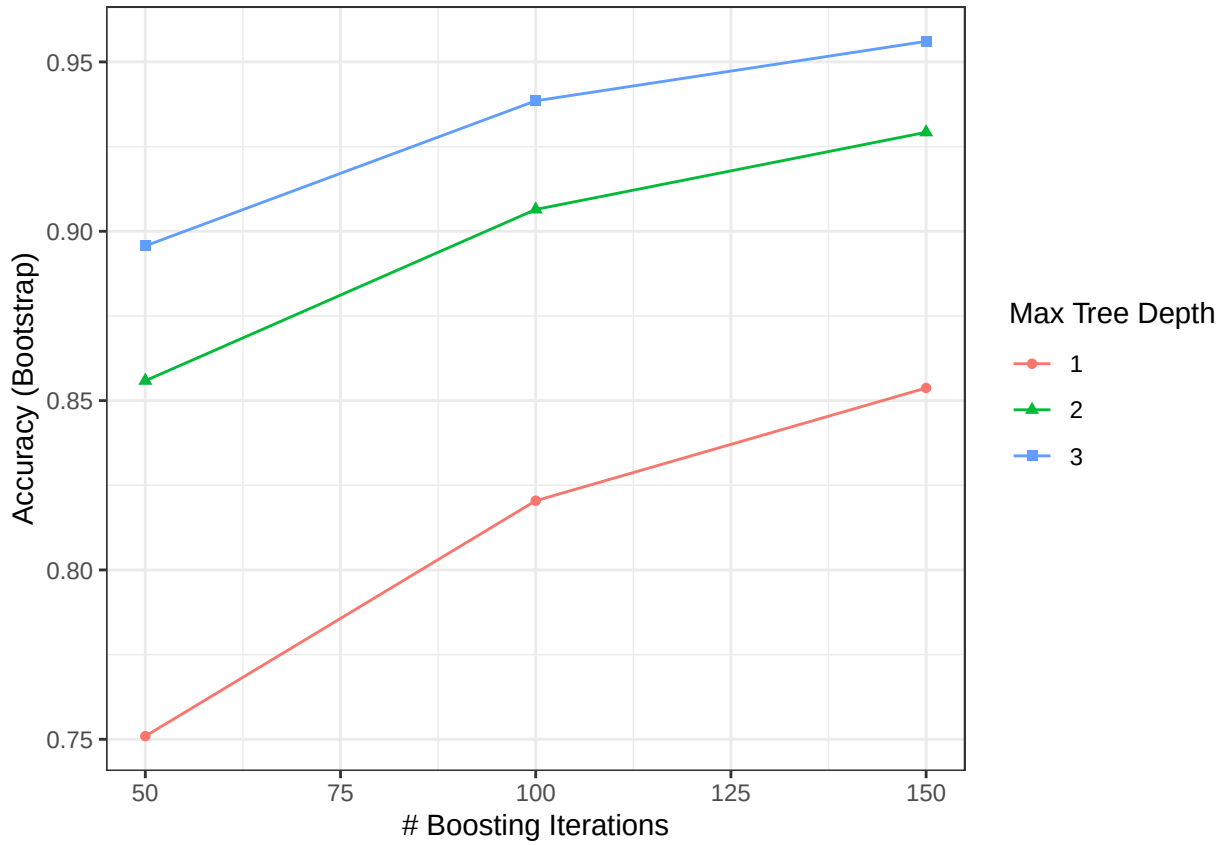
```
##          accuracy oos_error
## Tree        0.533     0.467
## RF          0.985     0.015
## GBM         0.957     0.043
## SVM         0.777     0.223
## Ensemble    0.790     0.210
```

We can see from the table above, Tree has the highest out of sample error followed by SVM and Ensemble. The best model in terms of accuracy and OOS error is random forest followed by GBM .

```
# plotting the models

library(ggplot2)
par(mfrow=c(1,1))
ggplot(modelFit1) + theme_bw()
```




```
ggplot(modelFit3) + theme_bw()
```



A simple plot between Complexity Parameter vs Accuracy (Cross-Validation) for decision tree model is shown above. As complexity parameter decreases the accuracy of the validation increases. For gradient boosting ,we can see that bootstrapping accuracy of the model increases as the iterations of boosting increases. We can also see that as the tree depth shifts upwards accuracy also increases.

Finally, let us apply the modelFit2 to the new testing set that does not have classe or some new data we haven't yet seen and predict the classe of each observations.

```
# prediction on testing dataset
predc = predict(modelFit2, newtest)
predc
```

```
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##  B  A  B  A  A  E  D  B  A  A  B  C  B  A  E  E  A  B  B  B
## Levels: A B C D E
```